Illinois Institute of Technology Department of Computer Science

Final Exam (Mostly Non-Cumulative)

CS 440— Programming Languages
Fall 2006
December 11, 2006 14:00–16:00

This is a **closed book** and **closed notes** exam.
You are **not** allowed to use calculators or computers during this exam.
Do **ALL** problems in this booklet. Read each question very carefully.
You may detach pages, but **you must return all pages of this exam.**

Name	
Email ID	@iit.edu

Do **not** place your social security number anywhere on this exam.

Problem	Points	Score
1	4	
2	4	
3	6	
4	6	
5	6	
6	4	
7	6	
8	6	
9	4	
10	6	
11	6	
12	6	
13	6	
14	6	
15	4	
16	6	
17	4	
Total	90	
Percent	100	

1 Higher Order Functions

Question 1) (4 points) Write the code for `map : ('a -> 'b) -> 'a list -> 'b list`.

Question 2) (4 points) Write the code for `fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

Question 3) (6 points) Using either `map` or `fold_right`, write the function `product xx : int list -> int`, which computes $\Pi_{i=0}^{n} a_i$ from the list $[a_0; a_1; \ldots; a_n]$. You may **not** use explicit recursion.

```
# let product xx = ...
val product : int list -> int = <fun>
# product [3;1;9];;
- : int = 27
```

Question 4) (6 points) Using either `map` or `fold_right`, write the function `decList xx : int list -> int list`, which decrements each element of a list. You may **not** use explicit recursion.

```
# let decList xx = ...
val decList : int list -> int = <fun>
# decList [2;4;6];;
- : int list = [1;3;5]
```

2 Prolog

Question 5) (6 points)
Consider the following Prolog code.

```
length([],0).
length([H|T],X) :- length(T,Y), X is Y + 1.
```

Show how to write the `prod` predicate, which is true when the second argument is the product of the elements of the first argument.

Question 6) (4 points) For what kind of applications is Prolog well-suited? Give an example.

Question 7) (6 points) Consider the following Prolog database.

```
connected(a,b).
connected(b,c).
connected(a,d).
connected(d,c).
etc...
```

This database represents the connections in a graph. Write a predicate `pathfrom(X,Y)` which is true when there exists a path from `X` to `Y` in the database. (Don't hard code it to this particular database; we may add other connections later.)

Question 8) (6 points) Consider the following Prolog database.

```
pet(puppy).
pet(tarantula).
adjective(happy).
adjective(hairy).
adjective(dangerous).
```

If you give the query

```
?- adjective(X), pet(Y).
```

how many results are returned?

Show how to insert a cut operator so that only one result will be returned.

What will that result be?

Question 9) (4 points)

Consider the following Prolog code.

```
valid(X) :- (pre1(X); pre2(X)), post(X).
```

The idea is that `X` is valid if it satisfies one of the two preconditions `pre1` and `pre2`; and then also satisfies the postcondition `post`. The predicate `post` is very expensive to run, however. Show how we can use the cut operator to make this query more efficient (without causing it to reject `X`'s that are valid, of course).

3 Recursion

Use recursion to write the following function. You do **not** have to use tail recursion, but you may do so if you want. You are **always** allowed to make helper functions.

Question 10) (6 points) Write a recursive function `sumonen : int -> int` that takes an integer n and returns the sum $\Sigma_{i=1}^{n} i$. If $n < 0$ simply return 0.

```
# let rec sumonen n = ...
val sumonen : int -> int = <fun>
# sumonen 10;;
val - : int = 55
```

Question 11) (6 points) Write the function `sum xx : int list -> int`, which returns the sum $\Sigma_{i=1}^{n} a_i$ from the list $[a_0; a_1; \ldots; a_n]$. Use tail recursion. You may write a helper function, or use standard library functions if you want.

4 State

Question 12) (6 points) Suppose you want to write an accumulator function. It keeps a local state, starting at 0, and then keeps a running total of the input. Here is a session:

```
# let acc = some secret stuff
val acc : int -> int = <fun>
# acc 2;;
- : int = 2
# acc 3;;
- : int = 5
# acc 5;;
- : int = 10
```

Give the code for `acc` that has the behavior above. You may not use a global variable.

Question 13) (6 points) State makes it difficult to use equational reasoning. Give an example, using the `acc` function above, that illustrates this.

Question 14) (6 points) Using local state, we can simulate objects. We also need a mechanism for simulating inheritance. Describe the message dispatcher model, and how it handles methods inherited from another object. (As a reminder, here is a counter object being called.)

```
# ct "inc" ();;
- : int = 1
# ct "reset" ();
- : int = 0
# ct "inc" ();;
- : int = 1
```

If you can describe how a "fast counter" might work, inheriting the reset method but overriding the inc method, that would answer the question nicely. You do not need to give code unless it helps your explanation.

Question 15) (4 points) We have commented that state makes equational reasoning difficult. But, we sometimes use it anyway. What is the advantage? (You will **NOT** get credit if you just say that it "makes programming easier," you need to be more specific than that.)

5 Unification

Question 16) (6 points) Solve the following unification problem. Show your work to allow for partial credit if something goes wrong. The Greek letters are the variables.

$$\{g(\alpha, x) = g(y, \beta), \quad h(\gamma, z) = h(f(\alpha), z)\}$$

Question 17) (4 points) Give an example of a language feature (from any language) which is implemented using unification.